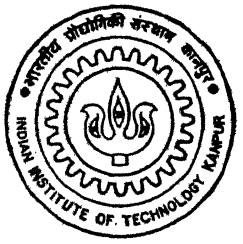
DATA DEPENDENCE TESTS FOR LOOP PARALLELIZATION

by

Sandeep Singhai



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY, 1995

DATA DEPENDENCE TESTS FOR LOOP PARALLELIZATION

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

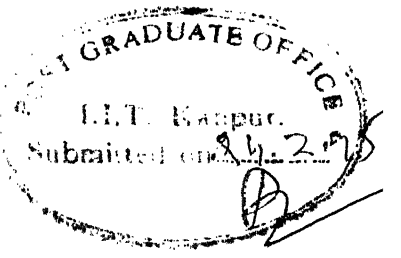
by

Sandeep Singhai

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February 1994



CERTIFICATE

This is to certify that the work contained in the thesis titled **Data Dependence Tests for Loop Parallelization** by **Sandeep Singhai**, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

(Dr. R. K. Ghosh)
Associate Professor,
Dept. of Computer Science & Engg.,
Indian Institute of Technology,
Kanpur.

(Dr. Sanjeev Kumar)
Associate Professor,
Dept. of Computer Science & Engg.,
Indian Institute of Technology,
Kanpur.

February 1994

Abstract

The data dependence analysis deals with determining if two references to an array can refer to the same element and if so, under what conditions. This information is used to determine allowable program transformations and optimizations. Hence, precise and efficient dependence tests are essential to the effectiveness of a restructuring compiler. Most of the existing dependence tests either provide inaccurate information or are applicable for a restricted class of problems. A dependence testing scheme integrating the existing data dependence tests into a unified framework is proposed to improve the efficiency of dependence analysis process. The approach followed is to take the different characteristics of a given problem instance into account and apply only those tests which are likely to be successful. An exact test, the GAtest, based on genetic algorithms is presented to address the issues of accurate yet fast analysis and extraction of fine grained parallelism. The GAtest is general in nature. It can handle triangular and trapezoidal search spaces and is applicable to coupled subscripts as well. The GAtest can enumerate all the solutions within the loop bounds. This information can be used to optimally execute a given loop nest on a parallel processor. Performance analysis shows that the GAtest can be used as a practical dependence test.

Index Terms — Compiler optimization, data dependence, genetic algorithms, restructuring compilers, parallelization.

Acknowledgments

I am grateful to my thesis supervisors, Dr. Sanjeev Kumar and Dr. R. K. Ghosh, for their constant support, encouragement and guidance throughout the course of this work. Dr. Sanjeev Kumar introduced me to the interesting field of Parallel Compilers and Dr. Ghosh encouraged me to explore the exciting domain of genetic algorithms. I also wish to thank them for teaching me the art of technical writing. It was really a joyful experience to observe how a *therefore* or an *obviously* could transform a set of disconnected sentences into a meaningful paragraph.

I thank Deepak Gupta for helping me in academic matters and for giving his nice company throughout my stay at IIT/K. I am grateful to the class of M. Tech. '93 for making the last one and half years, a memorable period of my life. In particular, I would like to thank Barot, Muralidhar, Pavan, Uzzal and Vidyut (in alphabetical order) for the help they extended to me and for all the fun that we had together.

Finally, I thank my parents and brothers for always encouraging me and for letting me make my own decisions.

Sandeep Singhai

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

As technology approaches certain physical limitations, parallelism seems to be the most promising alternative to satisfy the ever increasing demand for computational power. The parallelism may be realized by using multiple functional units or by employing more than one CPUs that work in mutual co-ordination.

Parallel computers may be classified into three architectural categories [8]:

- Pipeline Computers
- Array Processors
- Multiprocessor and Multicomputer Systems

Pipeline computers exploit temporal parallelism by overlapping the computations. Most of the pipeline computers use vector processing, where the same operation is repeated several times. Array processors exploit spatial parallelism by operating multiple arithmetic units in lock-step fashion. Multiprocessor systems usually contain two or more processors of equal capabilities, with the inter-processor communication being done through the shared memory. The multicomputer systems also contain two or more processors but they use message passing for inter-processor communication.

Several factors should be considered when designing high performance parallel computers [11]. Parallel algorithms, architecture and programming environments all play important roles on program performance. The investment in software for serial machines, however, is so high that it will be many years before parallel software dominates. The need to run serial software on vector or parallel machines without reprogramming gives rise to program restructuring. A restructuring compiler transforms a sequential program to take advantage of the architectural characteristics of a machine.

A restructuring compiler performs a number of transformations to extract the parallelism in the given sequential program. These transformations must preserve the flow correctness of the program. The flow correctness can be ascertained by ensuring that the dependence constraints on the memory references in the program are preserved. In general, data dependences give information about how data are defined and used in the program.

For example, consider the loop:

```
DO I = 1, 100
    S1 : B(I) = A(I) + C(I)
    S2 : A(I+2) = D(I)+E(I)
ENDDO
```

In the above loop, the value of `A(I+2)`, computed in iteration `I` $= i_0$, is used in statement S_1 in iteration `I` $= i_0 + 2$. If the loop is transformed such that processor P_0 is executing iteration `I` $= i_0$ and processor P_2, iteration `I` $= i_0 + 2$, then P_2 must wait until the value of `A(I+2)` has been computed by P_0 and is ready for use by P_2.

On the other hand, in the following code fragment, no location of arrays is read or written more than once. This allows the loop to be executed in any order.

```
DO I = 1, N
    S1 : A(2*I) = B(I) + K
    S2 : C(I) = A(2*I+1) + D(I)
ENDDO
```

In order to determine the validity of restructuring transformations, data dependence tests are devised to detect programs or loops whose semantics would be violated by the transformation. The problem of determining whether two occurrences of the same scalar result in a dependence is easy. Therefore, the various dependence tests mainly concentrate on dependence problems arising out of array references.

1.2 Data Dependence Concepts

The data dependence problem can be formulated as follows. Consider the following loop-nest:

```
DO I1 = L1, U1
    DO I2 = L2, U2
        ...
            ...
                DO In = Ln, Un
                    S1 : A(f1(I1, ... , In), ... , fm(I1, ... , In)) = ...
                    S2 : ... = A(g1(I1, ... , In), ... , gm(I1, ... , In))
                ENDDO
            ...
        ...
    ENDDO
ENDDO
```

The statement S_2 is dependent on S_1 if and only if there is an integer solution to the system of equations

$$\begin{aligned} f_1(i_1, \ldots, i_n) &= g_1(j_1, \ldots, j_n) \\ \ldots & \qquad \ldots \\ f_m(i_1, \ldots, i_n) &= g_m(j_1, \ldots, j_n) \end{aligned} \tag{1.1}$$

satisfying the constraints

$$L_k \leq i_k, j_k \leq U_k$$

such that $\mathbf{I} = (i_1, \ldots, i_n)$ is lexicographically less than or equal to $\mathbf{J} = (j_1, \ldots, j_n)$. Depending on the order of "define" and "use" of the array element causing a dependence, four cases are possible: *define-use*, *use-define*, *define-define* and *use-use*. The first case is called *flow dependence* and is denoted by $S\delta T$, where S and T are the two statements involved in the dependence. The second case is called *anti-dependence*, written $S\bar{\delta}T$. The third case is of *output dependence* and the symbol δ^o is used to represent it. The last ordering, *use-use*, is not much important because it does not require any synchronization.

Example 1.1 Consider the following loop.

```
DO I = 2, N-2
    S1 : A(I) = B(I)
    S2 : C(I) = A(I-1)
    S3 : A(I+1) = C(I+2)
ENDDO
```

In this case, $S_1\delta S_2$ because the array element `A(I)` assigned in iteration `I` $= i_0$ is fetched by statement S_2 in the next iteration of the loop. Similarly, $S_3\bar{\delta}S_2$ because `C(I+2)` used in iteration `I` $= i_1$ is defined by S_2 in iteration `I` $= i_1 + 2$. The pair of statements S_1 and S_3 give rise to output dependence $S_3\delta^o S_1$, because the array location `A(I+1)` written in iteration `I` $= i_2$ by statement S_3 is again assigned a value by statement S_1 in the next iteration.

The data dependence relations for a program segment are often modeled by graphs. A data dependence graph G is associated with a program fragment with

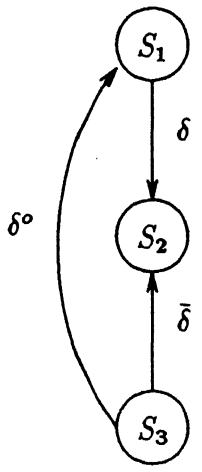


Figure 1.1: The data dependence graph for Example 1.1

s statements. The graph G has s nodes, one for each S_x $(1 \leq x \leq s)$. For each dependence relation between S_v and S_w, there is a corresponding arc in G from the node representing S_v to the node representing S_w. Figure 1.1 shows the data dependence graph (DDG) of the program fragment given in Example 1.1.

The dependence between two statements may be characterized using the distance and direction vectors [1] in the following way. A distance vector $\mathbf{D} = (d_1, \ldots, d_n)$ and a direction vector $\mathbf{S} = (s_1, \ldots, s_n)$ is associated with each pair of solution vectors $\mathbf{I}$ and $\mathbf{J}$ where $d_l = j_l - i_l$ and

$$s_l = \begin{cases} < & \text{if } j_l < i_l \\ = & \text{if } j_l = i_l \\ > & \text{if } j_l > i_l \end{cases}$$

The case when there is a dependence between S_1 and S_2 for the same iteration is relatively less important because such iterations do not require any synchronization. The dependences between different iterations of the loop are known as loop-carried dependences. The presence of a "=" element in the direction vector denotes loop independent dependence.

Empirical studies indicate that usually the array references are linear expressions over the iteration space [1]. Assuming that f_i and g_i are linear functions, the problem of dependence testing for a pair of array subscripts then reduces to the problem of

finding all solutions to systems of linear diophantine equation

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = a_0 \tag{1.2}$$

in a bounded space. This problem has been proved to be NP-complete [4]. Therefore, most of the practical dependence tests check for the existence of certain necessary conditions for an integer solution to exist. These tests are conservative in nature, i. e., they assume dependence unless it is explicitly ruled out by a violation of the necessary conditions. Hence they preserve the semantics but at the same time, the restructured program is not always optimally efficient. Such tests are known as inexact tests. On the other hand, exact tests detect dependence if and only if it exists.

1.3 Dependence Tests

There have been basically two approaches in dependence testing [13]. The first approach employs numerical methods while the second is based on checking the consistency of a set of inequalities and equalities. The restructuring compilers have traditionally employed two inexact dependence tests [1, 20] — the GCD test and Banerjee's inequalities. Both of these tests are based on the theory of diophantine equations and can be classified in the category of numerical algorithms. The GCD test predicts unconstrained integer solutions and Banerjee's test indicates the presence of constrained real solutions. But in many of the common cases, these tests declare false dependence. For example, when $gcd(a_1, \ldots, a_n) = 1$, GCD test always predicts dependence although it may not exist. Similarly, the presence of constrained real solution indicated by the Banerjee's test does not always imply that there is a constrained integer solution too.

The I test [10] combines the GCD and Banerjee's test. It generalizes the GCD test to interval equations and guarantees a performance at least as good as that of Banerjee's test. The I test may be applied even when sufficiently many limits are not known for Banerjee's test to be applied. However, it does not prove to be significantly better than the GCD and Banerjee's test when sufficiently many loop limits are known.

λ-test [13] is another approximate test. It is an extension of the Banerjee's test for the multidimensional case. λ-test extends the numerical methods to check for constrained real solutions to a given system of equations and inequalities. The Power test [22], a more precise algorithm, uses Banerjee's Generalized GCD test and an integer programming method, the Fourier Motzkin variable elimination. Unfortunately, the high cost of the Power test precludes its use in a practical system.

The single-index-exact test [1, 20] may be used if only one index variable appears in each of the subscripts. It can also enumerate all the solutions. The Delta test is an exact test for certain classes of subscript references [5]. It works by deriving constraints from single-index-variable subscripts and propagating them into other dimensions. The Delta test is very efficient at calculating exact distance and direction vectors for some commonly encountered array references.

The Omega test [15] is an exact dependence test of general nature. It combines new methods for eliminating equality constraints with an extension of Fourier-Motzkin variable elimination. The Omega test determines whether there is an integer solution to a given system of linear equalities and inequalities. Experimental results indicate that the time taken by the Omega test to analyze a problem is rarely more than twice the time required to scan the array subscripts and loop bounds [15]. This makes the Omega test suitable for use in practical systems.

1.4 Scope of the Thesis

The work reported in this thesis, addresses two issues related to data dependence analysis. First, integration of the existing dependence tests into a unified framework in order to improve the efficiency of the process of dependence analysis. Second, determination of complete and exact information about a given dependence problem.

Most algorithms for dependence analysis are effective only for a particular class of problems. Therefore, it is desirable to identify a set of tests which are most likely to extract the maximum parallelism for a given problem in a practical system. We propose an integrated approach, called the Zero test, to address this issue. The Zero test incorporates most of the important dependence tests [1, 10, 13, 15, 20]

into a framework. It helps in selecting one or more of the tests appropriate for a given problem instance. Intuitively, the Zero test achieves its goal by applying more complex dependence tests only for those problems that actually require them.

As mentioned in Section 1.2, almost all practical dependence tests either provide inaccurate information or are applicable for a restricted class of problems. In this thesis, we present the GAtest, an exact test of general nature. It introduces concepts from Genetic Algorithms (GAs) into framework of dependence analysis. Genetic algorithms are search techniques based on the principle of survival of the fittest with a structured, yet randomized information exchange [6]. Some other interesting applications and potentials of genetic algorithms can be found in [2, 3]. The interested readers may refer to [6, 7] for a detailed study of the subject.

The GAtest employs genetic algorithms to obtain an integer solution, if any, within the loop bounds. It can enumerate all the solutions which can be used to exploit the fine grained parallelism. When the dependence distance is not constant, this information allows us to optimally execute a given loop nest on a parallel processor. The GAtest can handle coupled subscripts and can test for direction vectors. It can also be applied to trapezoidal and triangular spaces apart from the rectangular spaces. Performance analysis demonstrates that the GAtest can be used as a practical dependence test.

The organization of the rest of this thesis is as follows. Chapter 2 describes how the Zero test uses the various properties of the loop-nests and of the dependence tests to improve the efficiency of dependence analysis. Chapter 3 presents the GAtest and gives a formal proof of correctness of the algorithm. We also discuss certain implementation issues and give empirical results comparing the GAtest with other methods. The conclusions and scope for further work are presented in Chapter 4. The example programs presented in this thesis are in a notation similar to Fortran. We often label statement with names like $S_1, S_2, \ldots$ for clarity.

Chapter 2

The Zero Test: An Integrated Approach for Dependence Testing

2.1 Introduction

The data dependence analysis has received great attention from the researchers and a number of tests have been developed [1, 5, 10, 13, 15, 20, 22]. These tests vary in generality, precision, complexity and execution cost. Therefore, in the context of practical design of a restructuring compiler, these tests should be evaluated appropriately in a unified framework. Clearly, such an evaluation has to be supported by exhaustive empirical results on known benchmark suites. This chapter addresses the above mentioned issue and evolves a decision model for application of various data dependence tests.

The Zero test, as it is called, selectively applies various data dependence tests depending on the characteristics of the given problem. The objective of this decision algorithm is to select only those tests which are likely to be efficient for the given dependence problem.

2.2 The Dependence Tests

The Zero test uses almost all the important data dependence tests available. In this section, these dependence tests have been outlined briefly. The objective is to identify different characteristics of these data dependence tests which can be used to help the decision process. Some important statistical results, if available, are also given.

2.2.1 The GCD and Banerjee's Test

Most of the compilers have traditionally relied on these two inexact tests [1, 20]. The GCD test is based on an elementary theorem of number theory which states that an integer solution to a linear equation $\sum_{i=1}^{n} a_i x_i = a_0$ exists if and only if $g = gcd(a_1, a_2, \ldots, a_n)$ is a divisor of a_0. The GCD test can indicate only unconstrained integer solutions. It has also been extended to determine if there is an integer solution to a given set of linear equations.

The Banerjee's test is based on intermediate value theorem and the theorem on bounds of linear functions. These theorem are proved in [1]. The Banerjee's test computes the bounds of linear expression $\sum_i a_i x_i$ and if a_0 satisfies these bounds, a constrained real solution can be predicted. The Banerjee's test has also been extended by Wolfe [20] to test for a given direction vector.

The bounds of $\sum_i a_i x_i$ in a rectangular space can be computed as follows. The lower bound B_{low} of $\sum_i a_i x_i$ occurs when

$$x_i = \begin{cases} l_i & \text{if } a_i \geq 0 \\ u_i & \text{otherwise} \end{cases}$$

Similarly, the upper bound B_{high} of $\sum_i a_i x_i$ occurs when

$$x_i = \begin{cases} u_i & \text{if } a_i \geq 0 \\ l_i & \text{otherwise} \end{cases}$$

This method has also been extended to compute bounds of a linear function in a trapezoidal space.

Although both of these tests are very simple and fast, they give conservative answers in many of the common cases. In the example given below, it can be shown that there is no integer solution within the loop bounds specified but the GCD test and Banerjee's test both predict dependence.

Example 2.1 Consider the following loop.

```
DO I = 1, 5
    S1 : A(3*I) = ...
    S2 : ...= A(8*I + 6)
ENDDO
```

The linear equation formed for this loop is $3x_1 - 8x_2 = 6$ with the constraint, $1 \leq x_1, x_2 \leq 5$. Since $gcd(3, -8) = 1$, which divides $a_0 = 6$, the GCD test indicates the presence of integer solutions. When the Banerjee's test is applied to the above loop, it computes the lower and upper bounds of the expression, $3x_1 - 8x_2$, as follows.

$$\begin{aligned} B_{low} &= min(3x_1 - 8x_2) = 3(1) - 8(5) = -37 \\ B_{high} &= max(3x_1 - 8x_2) = 3(5) - 8(1) = 7 \end{aligned}$$

Since $-37 \leq 6 \leq 7$, the Banerjee's test also cannot rule out the dependence.

2.2.2 The I test

The I test proposed by Kong *et al.* [10] is a refinement of a combination of the GCD and Banerjee's test. The I test generalizes the GCD test to the interval equation

$$\sum_{i=1}^{n} a_i x_i = [a_l, a_u]$$

This equation has an integer solution if and only if the following condition, known as the interval GCD test, is satisfied.

$$a_l \leq g\lceil a_l/g \rceil \leq a_u$$

where $g = gcd(a_1, a_2, \ldots, a_n)$. If the given diophantine equation satisfies the interval GCD test, an i such that $|a_i| \leq a_u - a_l + 1, 1 \leq i \leq n$ is chosen and a reduction is

made by computing new interval $[a'_l, a'_u]$ as follows.

$$a'_l = a_l - max(a_i x_i)$$
$$a'_u = a_u - min(a_i x_i)$$

The above process is repeated until interval GCD test fails or further reductions are not possible. The following example, reproduced from [10], illustrates the method used by the I test.

Example 2.2 Consider the equation

$$x_1 - 3x_2 + 7x_3 = 8$$

subject to the constraints, $1 \leq x_1 \leq 3$, $1 \leq x_2 \leq 2$ and $1 \leq x_3 \leq 4$. The initial interval equation for the above equation is

$$x_1 - 3x_2 + 7x_3 = [8, 8]$$

which satisfies the interval GCD test. At this stage, a reduction is made with $i = 1$ because $a_1 = 1 \leq 8-8+1$. The following interval equation is formed after computing new interval

$$-3x_2 + 7x_3 = [5, 7]$$

The interval GCD test is again satisfied since $gcd(-3, 7) = 1$ and $5 \leq 1\lceil 5/1 \rceil \leq 7$. Now, $i = 2$ is selected because $|a_2| = 3 \leq 7 - 5 + 1$. The new interval equation is

$$7x_3 = [8, 13]$$

which rules out dependence because $gcd(7) = 7$ and $8 \leq 8\lceil 8/7 \rceil \not\leq 13$. It should be noted that the I test could detect absence of integer solutions without using the bounds of x_3.

The I test is also a fast test and it has been shown [10] that its worst case complexity is BAN + $n \times$ GCD where

- BAN is the cost of a single Banerjee's test
- GCD is the cost of a single GCD test
- n is the number of variables in the given equation

2.2.3 The Single-Index-Exact Test

If only one index variable appears in each of the subscripts, the single-index-exact test may be applied. This test makes use of the following theorem [1]:

Theorem 2.1 *Let a, b and c be integers such that a and b are not both zero and let $g = \gcd(a, b)$. If (i_0, j_0) is a solution to the equation*

$$ai - bj = c$$

then

$$(i_0 + tb/g, j_0 + ta/g)$$

would also be a solution for any integral value of t.

The single-index-exact test works as follows. Let S and T denote the two statements being tested for possible dependence and let l_1, u_1 be the lower and upper bounds, respectively, of the given loop. Suppose that the two subscript expressions are $a_0 + ai$ and $b_0 + bj$, where a_0, b_0, a and b are integer constants. Let $c = b_0 - a_0$. Now, we need to solve the following diophantine equation

$$ai - bj = c \tag{2.1}$$

subject to the conditions

$$\begin{array}{lllllll} l_1 & \leq & i & \leq & j & \leq & u_1 \quad \text{for } S < T \\ l_1 & \leq & i & \leq & j-1 & \leq & u_1 \quad \text{for } T \leq S \end{array} \tag{2.2}$$

to decide if $S\delta T$ holds.

Kirch's algorithm [20] can be used to find a particular solution (i_0, j_0) to Equation 2.1. By Theorem 2.1, $(i_0 + tb/g, j_0 + ta/g)$ is a general solution to Equation 2.1 for any integral value of t. Furthermore, since every solution to the given equation must also satisfy the conditions in 2.2, following constraints are obtained.

$$\begin{array}{lllllll} l_1 & \leq & i_0 + tb/g & \leq & j_0 + ta/g & \leq & u_1 \quad \text{for } S < T \\ l_1 & \leq & i_0 + tb/g & \leq & j_0 + ta/g - 1 & \leq & u_1 \quad \text{for } T \leq S \end{array}$$

The lower and upper bounds of t, t_l and t_u, respectively, may be obtained from these constraints. A dependence exists if $\lfloor t_u \rfloor - \lceil t_l \rceil \geq 0$. The bounds on t may also be used to enumerate all the solutions.

Example 2.3 Consider the loop in Example 2.1. $S_1 \delta S_2$ holds if the equation $3i - 8j = 6$ has an integer solution with the constraint $1 \leq i \leq j \leq 5$. Applying Kirch's algorithm gives a particular solution, say $(10, 3)$, to the given equation. Since $g = gcd(3, -8) = 1$, $a = 3$ and $b = -8$, $(10 - 8t, 3 + 3t)$ gives the general solution. Subjecting the general solution to the constraint $1 \leq i \leq j \leq 5$ and simplyfying, we get the bounds, $t \leq 2/3$ and $t \geq 7/11$. Since $\lfloor 2/3 \rfloor - \lceil 7/11 \rceil = -1 \not\geq 0$, there are no integer solutions to the given equation and hence, $S_1 \delta S_2$ does not hold.

2.2.4 The Delta Test

The Delta test is an exact test for certain classes of subscript references. It works by deriving constraints from single-index-variable (SIV) subscripts and propagating them into other dimensions [5]. Here, we give a small example which illustrates the concept of constraint propagation. Let the given pair of array references be $A(i+2, i+j+1)$ and $A(i, i+j)$. An SIV test applied to the first dimension derives a dependence distance of 2 for index i. Now, this constraint can be propagated to the second subscript, eliminating the occurrences of i. The resulting subscript is $\langle j-1, j \rangle$ which gives a distance of -1 on loop j. Thus, we can conclude that a dependence exists with distance vector $(2, -1)$. The constraints may also be intersected to solve the dependence problem for all the subscripts simultaneously. If the result of the intersection is the empty set, data independence may be concluded.

2.2.5 The λ-test

The λ-test [13] is an approximate test which extends the Banerjee's test for multidimensional case. As described in Section 1.2, a dependence between two statements S_1 and S_2 exists if and only if conditions in Equation 1.1 are satisfied. The λ-test solves a less exact problem by finding a λ-tuple $(\lambda_1, \lambda_2, \ldots, \lambda_m)$ such that

$$\lambda_1(f_1(\mathbf{I}) - g_1(\mathbf{J})) + \ldots + \lambda_m(f_m(\mathbf{I}) - g_m(\mathbf{J})) = 0 \qquad (2.3)$$

is satisfied. It should be noted that there could be instances of $\mathbf{I}$ and $\mathbf{J}$ which satisfy Equation 2.3 but not 1.1. Simultaneous real valued solutions to the given subscript expressions exist if and only if the Banerjee's test indicates solutions for all the linear

combinations generated. If the λ-test succeeds in finding a λ-tuple which cannot satisfy Equation 2.3, the dependence can be ruled out.

Example 2.4 Consider the loop-nest given below.

```
DO I = 1, 50
    DO J = 2, 50
        A(I, I+1) = A(3*I+J+1, I+J)
    ENDDO
ENDDO
```

The following linear equations need to be solved to determine data dependence.

$$\begin{aligned} x_1 - 3x_3 - x_4 &= 1 \\ x_1 - x_3 - x_4 &= -1 \end{aligned} \tag{2.4}$$

with the constraints $1 \leq x_1, x_3 \leq 50$ and $2 \leq x_4 \leq 50$. If $(-1, 1)$ is chosen as the λ-tuple, we have

$$-(x_1 - 3x_3 - x_4) + (x_1 - x_3 - x_4) = -2$$

The above equation implies that $x_3 = -1$, which clearly violates the bounds $2 \leq x_3 \leq 50$. Hence, there is no dependence for the given pair of array references.

The precision of λ-test may be improved by applying single-index-exact test to the linear combinations generated. The λ-test may also be applied for triangular loops and it can test for direction vectors.

2.2.6 The Power Test

The Power test [22] uses extended GCD algorithm [1, 9] and the Fourier-Motzkin elimination. The extended GCD algorithm gives a general solution to a given system of linear equations. If a system of d equations in n variables is being solved, the general solution consists of $n - d + 1$ free variables. The Power test determines the bounds of each free variable. A violation of any of the bounds rules out the possibility of dependence.

Example 2.5 We apply the Power test to the system of equations 2.4 given in Example 2.4. The extended GCD algorithm gives the general solution

$$(x_1, x_3, x_4) = (t_3 - 2, -1, t_3)$$

Since the value of x_3 violates the bound $1 \leq x_3 \leq 50$, data independence may be concluded.

2.2.7 The Omega Test

The Omega test, an exact dependence test, combines new methods for eliminating equality constraints with an extension of Fourier-Motzkin elimination to integer programming [15]. It determines whether there is an integer solution to a given system of equalities and inequalities. The Omega test works in the following way. Given a set of equality and inequality constraints, first, the equality constraints are eliminated, producing an equivalent set of inequality constraints. Next, all the redundant inequalities are eliminated and all the pairs of tight inequality constraints are replaced by corresponding equality constraints. These new equality constraints are handled in the same way as the equality constraints in the original problem. If there are any contradictory inequality constraints, the problem has no solutions. Otherwise, the Omega test projects constraints in n-dimension to $(n-1)$-dimensions. This way, the variables are eliminated until a single variable is left for which the existence of integer solutions is easy to check. Intuitively, the projection of a set of constraints is a shadow of the object defined by the set of constraints. For example, projecting $\{0 \leq x \leq 5;\ y \leq x \leq 5y\}$ onto x gives $\{2 \leq x \leq 5\}$.

The Omega test is very general in nature and it can efficiently produce a set of constraints that accurately describe all possible dependence distance vectors. It may also be used for obtaining other useful information. For example, it can determine if one region is a subset of another. The Omega test has also been extended to handle killing, covering and terminating dependences and to refine dependence distances [16].

2.3 An Integrated Approach

The motivation for the Zero test comes from the fact that a number of data dependence tests are available but there has been no attempt to integrate them into a unified framework. Since many of the tests are very costly, it is not possible to apply all of them in a practical system. Most of the existing restructuring compilers rely on very simple inexact tests. Another possibility is to take interactive feedback from the user to choose a specific test for application. The interactive feedback from user in some sense coerces the compiler to capture intrinsic properties of the given problem in the context of data dependence.

It was realized that the process of data dependence analysis can be made more efficient if the properties of the given problem instance are taken into account to select one or more appropriate tests for application. The selected tests should be more likely to expose the maximum parallelism at the minimum possible cost. There are two important issues related to the approach of selectively applying dependence tests. First, the execution cost of the various algorithms for different problem instances varies considerably. While there have been some studies in this regard, the subject has not received comprehensive treatment. Second, most of the tests are inexact and it is not well understood as under what conditions these tests provide correct answers. Hence, the statistical observations play an important role in determining the efficacy of dependence tests.

2.3.1 Characteristics of the Array References

The various properties of the given array references that should be considered, to improve the efficiency of dependence analysis, are as follows.

The number of subscripts. The applicability and performance of some of the data dependence tests depend on the number of subscripts in the given array references. For example, single-index-exact test may be applied for single dimension references since such subscript expressions have only one index variable.

Separable and coupled subscripts. If the same index variable occurs in two different subscript expressions, the latter are called coupled subscripts [13]. On

the other hand, the separable subscripts have index variables which occur in one subscript only. The coupled subscripts require special treatment because independent constrained integer solutions for each of the subscripts do not guarantee that there would be a solution for the array reference as a whole. Consider the following example, reproduced from [13]:

```
N = 50
DO I = 1, N
    DO J = 2, N
        S1 : A(2*I+3*J+N, 3*I+J+N-1) = ...
        S2 : ...= A(I-J+N+1, 2*I-J+N-2)
    ENDDO
ENDDO
```

We can show that there is no integer solution which satisfies both the subscripts simultaneously in this example. But there are solutions for each of the subscripts independently. If we let I $\equiv x_1$, J $\equiv x_2$ in statement S_1 and I $\equiv x_3$, J $\equiv x_4$ in statement S_2 then $(x_1, x_2, x_3, x_4) = (1, 2, 9, 2)$ is a solution to the first subscript and $(x_1, x_2, x_3, x_4) = (1, 2, 4, 2)$ is a solution to the second subscript.

The subscript by subscript dependence tests, e.g., the I test, are not likely to be very effective in providing exact answer for coupled subscripts.

The coefficients of the index variables. Some inexact tests, e. g., the GCD test and I test, give conservative answers whenever the coefficients assume certain values. Some other inexact tests, for example, the λ-test, provide exact answers if the coefficients meet some specific conditions. These features would be described in more detail in the course of presentation.

Presence of symbolic variables. The subscript expressions may contain an unknown variable (i. e. a non-index variable with an unknown value). The unknown variables may also be dummy parameters of subroutines. In the following loop, we would like to determine that there are no dependences even if we had no information about the value of N:

```
DO I = 1, N
    A(I + N) = A(I)
ENDDO
```

Unfortunately, most of the dependence tests cannot be extended to allow the symbolic variables. However, integer programming dependence analysis methods can handle loop-invariant symbolic variables by adding them as extra constraints.

Loop limits. Certain tests, e.g., the Banerjee's test, are applicable only when the loop limits are known. On the other hand, more sophisticated algorithms, for example, the I test, can be applied even when the loop bounds are unknown.

2.3.2 Properties of the Dependence Tests

As mentioned earlier, the applicability and performance of various dependence tests should be taken into account to improve the efficiency of dependence analysis phase. In this section, we briefly describe the properties of the various tests and give statistical results, if available. Our emphasis is on stating the results which help the Zero test in selecting one or more tests to be applied. Interested readers may refer to the corresponding references for a fuller treatment.

Property 2.1 The GCD and Banerjee's tests provide inexact answers in many of the common cases. For example, the GCD test assumes dependence whenever any of the coefficients a_i has the absolute value of 1 or in general, whenever $gcd(a_1, a_2, \ldots, a_n) = 1$. It has been empirically observed that a large percentage of coefficients of loop indices in subscript expressions satisfy this condition [17], reducing the applicability of the GCD test. Similarly, the constrained real solutions predicted by the Banerjee's test do not imply that there is a constrained integer solution too.

These tests may not provide exact information but their execution cost is so low that their use contributes very little to the total execution time taken by the dependence analysis.

Property 2.2 The I test is also a fast test and it provides benefit over the Banerjee's test, because it is applicable even when sufficiently many loop limits are not known for the Banerjee's test to be applied. But the I test does not prove to be significantly better than the GCD test and Banerjee's test when sufficiently many loop limits are known. Moreover, the I test is applicable only when one of the coefficients a_i in the given linear diophantine equation satisfies the condition $|a_i| \leq 1$.

Property 2.3 Many of the complex tests subsume the single-index-exact test. But the high percentage of single dimension array references demands the simpler single-index-exact test to be applied. According to an empirical study reported in [17], the single dimension subscript expressions are found to be around 55% of all the array references in real programs.

Property 2.4 The Delta test is very efficient at calculating exact distance and direction vectors for some frequently occurring array references. Goff, *et al.* [5] give detailed empirical results demonstrating that the Delta test is applicable for scientific programs.

Property 2.5 Since the λ-test checks for simultaneous solution to the subscript expressions, it is suited for coupled subscripts. Shen, *et al.* [17] found 44% of the two dimensional references to be coupled in a study of various benchmark programs. It was also observed that the number of coupled subscripts in higher dimensional loops was negligible. The λ-test proves to be quite effective in practice. Empirical studies show that the λ-test usually increases the cost of dependence testing by a factor of two or less [13]. Li [12] has shown that the λ-test is exact for two coupled dimensions if the coefficients of index variables are all 1, 0 or -1.

Property 2.6 The Power test is more accurate than the Delta test, the λ-test and it subsumes the single-index-exact test. Unfortunately, the high cost of the Power test precludes its use in practice. In fact, Triolet [19] has shown that using Fourier-Motzkin elimination may take from 22 to 28 times longer than the conventional dependence testing.

Property 2.7 The Omega test is the most comprehensive test. It has exponential worst-case time complexity but in many situations, it has low-order polynomial time complexity. Experimental results indicate that the time taken by the Omega test to analyze a problem is rarely more than twice the time required to scan the array subscripts and loop bounds [15]. The general nature and low execution cost of the Omega test make it a practical method for performing dependence analysis for problems encountered in restructuring Fortran code.

Property 2.8 The Omega test can handle the symbolic variables and some types of `min` and `max` functions in the loop bounds. The symbolic variables are treated as additional variables to the integer programming problem. The Power test can also tackle some of the subscript expressions with symbolic variables but the Omega test is more comprehensive in this regard.

2.3.3 The Zero Test

The Zero test makes a decision based on the above mentioned characteristics of the array references and the properties of the various dependence tests. It applies one or more tests in succession and as soon as any of the tests rules out dependence, the analysis stops. If the dependence is indicated by an exact test, we need not apply other tests because they would also report the same. On the other hand, if an inexact test indicates dependence then the Zero test applies more precise algorithms, because the inexact tests may give false positive results.

The block diagram of the Zero test is shown in Figure 2.1. The diagram shows the action taken upon the indication of dependence only, through "dep" edges. Whenever data independence is established, the analysis is complete for the given array references.

The rationale behind the structure of the Zero test and the various properties used are described in Table 2.1.

The objective of the Zero test is to reduce the total number of tests being applied by selecting only those tests which are likely to be successful. For example, if the λ-test is not able to conclude data independence, the Zero test does not apply the GCD and Banerjee's test which are less accurate for the coupled subscripts. As can

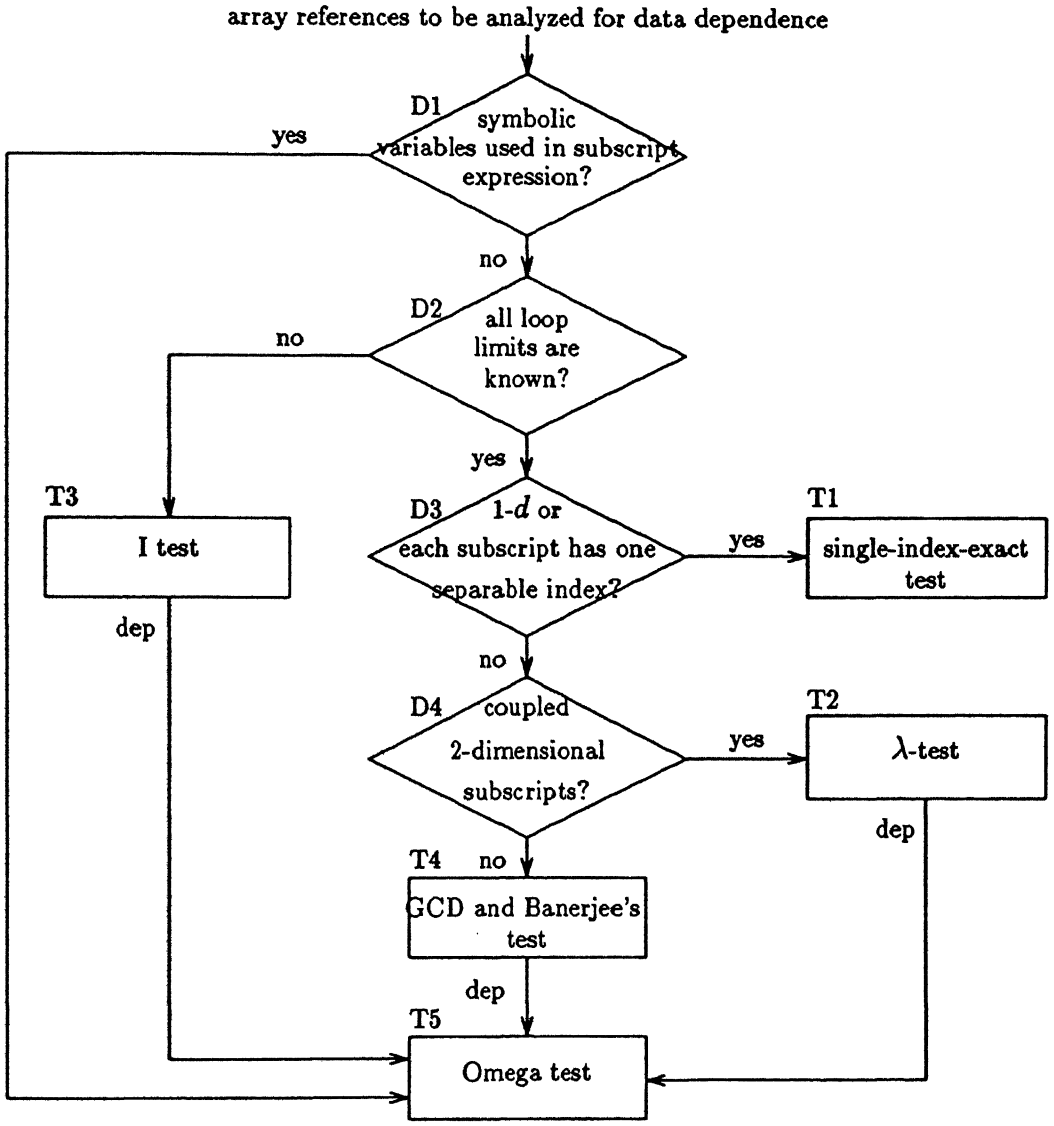


Figure 2.1: Block diagram of the Zero test

Block	Property used	comments
T1	2.3	Includes some of the optimizations suggested by the Delta test.
T2	2.5	Comparatively simple and fast for two dimensional cases.
T3	2.2	Useful only when some of the bounds are not known and at least one of the coefficients satisfies the condition $\|a_i\| \leq 1$.
T4	2.1	Very low execution cost.
T5	2.7, 2.8	Applied as a last resort. Not needed if very complex array references are not expected in the input programs.

Table 2.1: The various properties used by the Zero test

be noted from Figure 2.1, the number of tests applied to any given problem would be at the most 3. The worst case occurs when the GCD and Banerjee's tests fail and then the Omega test is applied. Even in this case, the execution cost would not be very high because the GCD and Banerjee's tests are not expensive.

The Zero test does not use the Power and Delta tests because of the following reasons. (1) The Power test is a very expensive test (Property 2.6). On the other hand, the Omega test is not only applicable for all those cases which the Power test can handle, it also gives a better performance. Therefore, there is no need to apply the Power test. (2) The Delta test is exact only for a restricted class of problems and it relies on expensive methods to handle the general case. The Zero test makes use of some of the optimizations suggested by the Delta test by including them in the single-index-exact test.

The data dependence analysis using the Zero test may be further optimized by introducing some details in the implementation. For example, if the coefficients of the index variables in the given problem are all 1, 0 or -1, then the Omega test need not be applied even if the λ-test indicates dependence. Because in this case, the λ-test gives exact answer.

Some other changes are possible depending on the requirements of the restructuring phase. For example, if the dependence distances are necessarily required then the λ-test should not be applied because it can determine direction vectors only. Moreover, some different properties of the loop-nests, e. g., range of the index

variables, may also be considered, if appropriate for a new dependence test.

2.4 Experimental Results

The Zero test as described in the previous section is a very general scheme. The various statistical results used in the design of the Zero test are representative of the array references usually encountered in scientific programs. But it should be emphasized that the structure of an integrated approach for dependence analysis depends on the particular applications for which the analysis is being done.

The Zero test was implemented in the TINY tool [21], a research tool for dependence analysis and loop restructuring. We analyzed the performance of the Zero test for the NASA NAS test suite and all the `tiny` programs supplied with the TINY tool. Some other small programs, representative of the array references usually encountered, were also used. These representative programs were taken from the relevant literature [1, 20]. Most of the programs, used for experiments, implement some numerical algorithms. For example, vpenta is a routine in the NASA NAS test suite which inverts three pentadiagonals simultaneously. This operation is commonly employed to solve partial differential equations. `relax` is another program supplied with the TINY tool. It implements the relaxation algorithm for a two dimensional grid.

The objective of the experiments was to determine the effectiveness of the Zero test and to tailor its structure appropriately, if required. Some important points about the framework used are as follows.

1. The analysis of array references that have constant values for the same pair of subscripts (e.g., `A(I,2)` and `A(I, 3)`) involves a simple comparison of the two constants. If they are not equal, the dependence is ruled out otherwise other appropriate tests are applied. In order to reduce the analysis time, every array reference pair is subjected to this simple test before any other tests are applied.

2. The single-index-exact test is applied not only to single dimension subscripts

Dimensions	No. of references
1	449
2	1292
3	1692
4	462

(a)

Coefficient	No. of terms
0	49122
±1	12956
other	182

(b)

Table 2.2: The characteristics of the array references

but also to multiple dimension separable subscripts. However, the loop indices used in the pair of the subscript expressions should be identical.

3. The version of Banerjee's test used for dependence testing, also computes the direction vectors.

We examined 3895 pairs of array references during the course of experimentation. Table 2.2 gives the characteristics of the array references analyzed. The number of array subscripts ranged from one to four. However, the majority of the array references had either two or three subscripts.

The enclosing loop-nest of the given pair of array references is considered to determine the coefficients of the loop indices in a subscript expression. Let a pair of array references be enclosed in a loop-nest of depth 3 and the loops be indexed by I, J and K, respectively. Now, the coefficients of I, J and K in the subscript expressions contribute to the respective counts. For example, absence of J in a subscript expression is considered as a case of coefficient 0. This way, all the subscripts of the given array references are considered one by one. The following example further illustrates how the counting is done:

```
DO I = 1, 100
    DO J = 1, 100
        S1 : A(I) = A(2*I+J)
    ENDDO
ENDDO
```

The enclosing loop indices of the pair of array referernces in the statement S_1 are I and J. The subscript expression of A(I) has one loop index, i.e. I, while J is

Dependence test	Usage frequency	Success Rate	
Constant value	2257	1575	69.78%
GCD and Banerjee's test	1443	52	3.60%
I test	17	7	41.17%
Single-index-exact test	1817	420	23.11%
λ-test	295	5	1.69%
Omega test	1218	270	22.16%

Table 2.3: The success rate of various dependence tests

absent. Therefore, it contributes one each to the counts for 0 and ± 1. Similarly, `A(2*I+J)` contributes one each to the counts of ± 1 and *other*. It can be noted from Table 2.2(b) that almost all the coefficients were either 0 or ± 1 in the programs used for experimentation.

In the array references analyzed, 524 subscript expressions were found to be non-linear. The non-linearity of most of these expressions can be attributed to the use of dummy parameters of subroutines. The value of such parameters can be made known to the dependence tests through user assertions. Normally, such assertions merely help the dependence tests to eliminate the same symbolic term in the subscript expression and do not affect the outcome of the test [17]. In our implementation, the user assertions were not used and all the subscripts containing symbolic variables were considered to be non-linear.

Out of 3895 pairs of array references analyzed, 2329 were found to have no data dependence by the Zero test. Table 2.3 gives the measured results. The usage frequency of a test is the number of times the test was applied. It should be noted that application of a subscript by subscript test to every subscript contributes to its usage frequency. If a subscript by subscript test concludes data independence for a particular subscript, the remaining subscripts are not tested. On the other hand, the tests which consider the array references as a whole, e.g. the λ-test, are applied only once to the array references, irrespective of the number of subscripts.

The success rate of a dependence test refers to the rate a particular test method concludes data independence. If a dependence test detects independence, success rate is counted only for this test although other tests could also potentially detect the independence.

The Omega test is the only expensive method used by the Zero test. It is obvious that the Zero test should employ the Omega test as sparingly as possible. Table 2.3 shows that the Omega test was applied 1218 times out of 3895. Other cases were identified as simple and less expensive methods were employed to solve them. The reduction in the number of times the Omega test had to be applied is significant, keeping in mind that the array references having non-linear subscripts are handled only by the Omega test. Furthermore, the success rate of the Omega test is adversely affected due to the following reason. Many of the array references, given to the Omega test, have already been conservatively declared to be dependent by other inexact tests. Hence, the probability of such references being dependent is comparatively high. In such cases, all the Omega test can do is to *confirm* the *tentative* result indicated by the inexact tests.

In view of the above observations, it can be concluded that the Zero test was able to meet its goal of identifying simpler cases and applying less expensive methods to solve them. The experimental results also suggest that the GCD and Banerjee's tests, which are traditionally employed by the restructuring compilers, are able to extract only a small fraction of the parallelism present in the programs.

Chapter 3

GAtest: An Exact Data Dependence Test

3.1 Introduction

Holland [7] proposed genetic algorithms in the early 1970s as computer programs that mimic the evolutionary processes in nature. Genetic algorithms manipulate a population of potential solutions to a search problem. Specifically, they operate on encoded representations of the solutions, equivalent to the genetic material of individuals in nature. Each solution is associated with a fitness value that reflects how good it is, compared with other solutions in the population. The various genetic operators, e. g., crossover, applied to the population result in probabilistic and useful information exchange between the different encoded solutions. These genetic operators, coupled with the *survival of the fittest* strategy, are employed to locate better solutions and ultimately, to solve the search (optimization) problem.

The feasibility of using genetic algorithms for testing dependence was first indicated in [18]. The solution as outlined there could provide only "yes" or "no" answer to the question of dependence between pairs of array references. However, subsequent restructuring of programs requires more information, e. g., direction vector, to be provided for efficient parallel execution of the programs. There are several other important aspects which were either ignored or addressed vaguely in

[18]. These include:

- using various genetic operators for improving efficiency of the GAtest.
- proving correctness of the GAtest in a theoretical framework.
- benchmarking GAtest against other well known dependence tests.

In this chapter, we present the results of a comprehensive investigation into all the above mentioned aspects of the GAtest. Such a comprehensive evaluation helps to determine whether the GAtest can provide an useful alternative to address the problem of dependence testing.

In its simplest form, GAtest can be used to determine the existence of dependence. The GAtest may also be employed to enumerate all the solution vectors and to determine the direction vectors. The solution enumeration helps in extracting the fine grained parallelism. Furthermore, the GAtest can be used to handle coupled subscripts by finding intersection of the solution sets of all the subscripts. We also suggest some simple strategies to solve the dependence problem for trapezoidal and triangular spaces. Statistical results, reported in Section 3.7, demonstrate that the GAtest can be used as a practical data dependence test.

3.2 GAtest — An Exact Data Dependence Test

The dependence problem may also be reframed as follows. In the search space defined by the loop bounds, we want to search an integer point $(x_1, x_2, \ldots, x_n)$, called the optimal point which satisfies the linear diophantine equation

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = a_0 \tag{3.1}$$

If we associate a fitness value $-|\sum_i a_ix_i - a_0|$ with every integer point $(x_1, x_2, \ldots, x_n)$ in the search space, then our problem is to obtain one of the fittest points in the search space, if it exists.

The GAtest uses determines an integer solution to the given diophantine equation as follows.

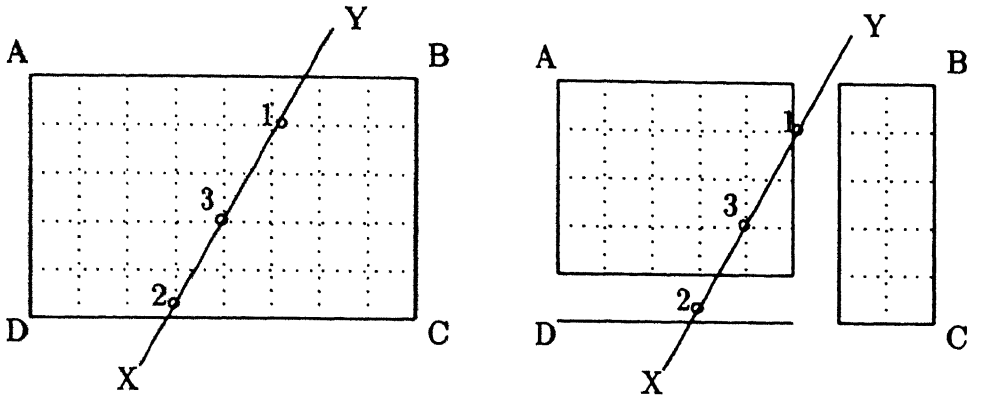


Figure 3.1: GAtest for rectangular search space

1. Apply Banerjee's test. If dependence is ruled out, there is no integer solution.

2. Use genetic algorithms to obtain a semi-integer[1] solution **xp** where **xp** = $(xp_1, xp_2, \ldots, xp_n)$.

3. If the semi-integer solution obtained in Step 2 has all the integer elements $xp_i\,(1 \leq i \leq n)$ then **xp** is the solution vector else divide the search space into two and apply GAtest to both the resulting spaces. The division of the search space is done by taking the floor and ceiling of the non-integer element of the vector **xp**.

The Figure 3.1 illustrates the idea behind GAtest for the simple case of a linear diophantine equation involving two variables: $a_1x_1 + a_2x_2 = a_0$. Let the rectangular space depicted in the figure denote the constrained space corresponding to the loop bounds. When the Banerjee's test is affirmative, the existence of a real solution is indicated. It has been proved later in Lemma 3.1 that Banerjee's test also provides necessary and sufficient conditions for the existence of a semi-integer solution. Some semi-integer solutions have been marked in the figure. Suppose, the GAtest finds the first solution at point labeled 1 in Figure 3.1. If this is a semi-integer but not integer solution, GAtest follows the divide and conquer strategy to partition the search space into two by taking the floor and ceiling of the non-integer element. Further applications of the GAtest result in the subspaces as shown in the figure.

[1]In Equation 3.1, **x** is a semi-integer solution if at most one $x_i, 1 \leq i \leq n$ is non-integer.

Algorithms 3.1 and 3.2 give a formal description of the GAtest.

Algorithm 3.1: GAtest

```
GAtest(l, u, a):boolean
begin
    if Banerjee_test(l, u, a) = true then begin
        x ← get_semi-integer_soln(l, u, a)
        if x_i (∀i) is integer then
            GAtest ← true
        else begin
            let x_i be the non-integer element of x
            tl ← l
            tu ← u
            tl_i ← ⌈x_i⌉
            tu_i ← ⌊x_i⌋
            GAtest ← GAtest(l, tu, a) ∨ GAtest(tl, u, a)
        end
    end else
        GAtest ← false
end
```

Algorithm 3.2: Algorithm to find a semi-integer solution

```
get_semi-integer_soln(l, u, a):semi-integer vector
begin
    POP ← initialize_pop(l, u)
    repeat
        apply_genetic_operator(POP)
        x ← select_a_vector(POP)
        j ← random_no_in_the_range(1, n)
        x_j ← (a_0 − Σ_i^{i≠j} a_i x_i)/a_j
        if (l_j ≤ x_j ≤ u_j) then begin
            get_semi-integer_soln ← x
            return
        end
        if x_j < l_j then
            x_j ← l_j
        else
            x_j ← u_j
        replace_worst_fit_vector_in_pop_by(x)
    forever
end
```

3.2.1 Correctness of GAtest

The correctness of the GAtest is proved by establishing that

1. The search space to which GAtest is applied contains at least one semi-integer solution.

2. Algorithm *get_semi-integer_soln* always returns a semi-integer solution in the search space if there is any.

Lemma 3.1 *Given an n-dimensional rectangular space $\Re$ and a diophantine equation $\sum_{i=1}^{n} a_i x_i = a_0$, Banerjee's test is positive iff $\sum_i a_i xp_i = a_0$ for some semi-integer vector* $\mathrm{xp} = (xp_1, \ldots, xp_n)$ *such that* $\mathrm{xp} \in \Re$.

Proof: The 'if part' is trivial because a semi-integer solution is a real solution.

The proof for 'only if' part can be derived from Theorems 4.1.1 and 4.2.3 of [1] in the following way. Let the vectors $A_1, A_2, \ldots, A_{2^n}$ represent the vertices of the n-dimensional rectangular space $\Re$. By Theorem 4.2.3 of [1], the bounds of the expression $\sum_i a_i x_i$ occur at these vertices. Let $val(A_l)$ $(1 \leq l \leq 2^n)$ represent the value of $\sum_i a_i x_i$ for vector A_l and the vectors corresponding to the minimum and maximum values of $\sum_i a_i x_i$ be A_i and A_j, respectively. That is, $val(A_i) \leq \sum_i a_i x_i \leq val(A_j)$. Now, consider a vector A_k, $1 \leq k \leq 2^n$. If $val(A_k) \leq a_0$ then $val(A_k) \leq a_0 \leq val(A_j)$. By Theorem 4.1.1 of [1], there exists a vector $\mathbf{X} \in \Re$ such that $A_k \leq \mathbf{X} \leq A_j$ which satisfies the equation $\sum_i a_i x_i = a_0$. If A_k and A_j differ only in at the most one component then $\mathbf{X}$ would be a semi-integer solution. If this is not the case then a vertex A_m may be chosen such that $A_k < A_m < A_j$ and the same reasoning can be applied recursively.

On the other hand, if $val(A_k) > a_0$, then $val(A_i) \leq a_0 \leq val(A_k)$. Again, there exists a vector $\mathbf{X} \in \Re$, such that $A_i \leq \mathbf{X} \leq A_k$, which satisfies the given diophantine equation. We follow the same reasoning as outlined before and ultimately, a vector $\mathbf{X}$ $(A_m \leq \mathbf{X} \leq A_n)$ is obtained, such that A_m and A_n differ only in at the most one dimension. ∎

Lemma 3.2 *Given a solution vector* $\mathbf{X}$, *the quantity* $|\sum_i a_i x_i - a_0|$ *decreases or remains stationary as the Algorithm 3.2 progresses.*

Proof: Let $a = a_j, x_{old} = x_j$ $(1 \leq j \leq n)$. Express $\sum_i a_i x_i$ as $Ax + ax_{old}$ where $Ax = \sum_i^{i \neq j} a_i x_i$. If $ax_{old} + Ax = a_0$, we already have a solution, i. e. $|\sum_i a_i x_i - a_0| = 0$ and we are done. Let $x_{new} = (a_0 - Ax)/a$.

case i. Let $ax_{old} + Ax > a_0$. If $a > 0$ then $x_{old} > (a_0 - Ax)/a$. That is $x_{new} < x_{old}$. Now, if $l_j \leq x_{new} \leq u_j$, it defines a solution. Otherwise, x_{new} saturates to $l_j > (a_0 - Ax)/a$ and the inequality $(ax_{old} + Ax - a_0) \geq (ax_{new} + Ax - a_0) > 0$ holds. On the other hand, if $a < 0$ then $x_{old} < (a_0 - Ax)/a$. This implies $x_{new} > x_{old}$. In case

$l_j \leq x_{new} \leq u_j$, we have a solution. Otherwise, x_{new} saturates to $u_j < (a_0 - Ax)/a$ and therefore, $(ax_{old} + Ax - a_0) \geq (ax_{new} + Ax - a_0) > 0$.

case ii. Let $ax_{old} + Ax < a_0$. In this case too similar arguments can be forwarded to prove that either a semi-integer solution is obtained or $(ax_{old} + Ax - a_0) \leq (ax_{new} + Ax - a_0) < 0$. ∎

Theorem 3.1 *Algorithm 3.2 always returns a semi-integer solution.*

Proof: Note that, Algorithm 3.1 applies the Algorithm 3.2 only when the Banerjee's test is true and in this case, Lemma 3.1 guarantees the presence of semi-integer solution(s).

By Lemma 3.2, the fitness value of a vector **X** fails to improve only in the case when x_j has been saturated to one of the limits l_j or u_j and the same j is chosen in the next iteration too. Since the value of j is being selected randomly, a different value of j would be chosen eventually and the algorithm would converge, returning a semi-integer solution. ∎

3.3 GAtest for Trapezoidal Spaces

Algorithm 3.1 solves the dependence problem for the rectangular search space, i. e., constant loop limits. However, In general, the loop bounds at level i may be linear functions of the index variables at levels 1 to $(i-1)$. The trapezoidal search space corresponding to such loop bounds can be characterized by

$$l_{i0} + \sum_{j=1}^{i-1} l_{ij}x_j \leq x_i \leq u_{i0} + \sum_{j=1}^{i-1} u_{ij}x_j \text{ for all } i = 1, \ldots, n$$

The divide and conquer strategy explained in Section 3.2 would not be appropriate for trapezoidal spaces. The reason is that the partitions of a trapezoidal space using Algorithm 3.1 would no longer be trapezoidal. Of course, a non-trapezoidal space could be partitioned into trapezoidal subspaces but this approach proves to be too expensive and complex to be implemented.

We propose an approach to handle trapezoidal spaces which retains the simplicity of GAtest. Consider the two dimensional trapezoidal space depicted in Figure 3.2. We observe the following:

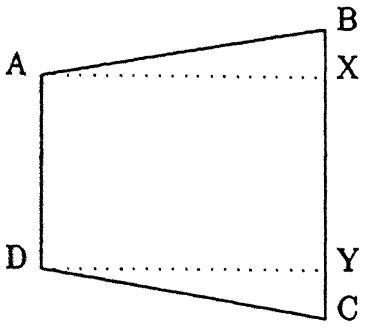


Figure 3.2: GAtest for trapezoidal spaces

1. If a semi-integer solution lies inside the rectangle AXYD, then the two partitions of the space ABCD would also be trapezoidal. In general, such an enclosed rectangular space can be found from the given lower and upper bounds as follows. For each dimension $i(1 \leq i \leq n)$, we compute

$$\begin{aligned} lmax_i &= max(l_i) \\ umin_i &= min(u_i) \end{aligned}$$

   For the i^{th} dimension, $lmax_i$ and $umin_i$ give the bounds for the enclosed rectangular space.

2. For any semi-integer solution in the given trapezoidal space, if the non-integer element corresponds to a dimension with constant bounds (in Figure 3.2, X-dimension) then also the resulting subspaces would be trapezoidal.

The same reasoning can be easily extended to n-dimensional space. In light of the above observations, the splits of a trapezoidal space can be obtained as given by the following scheme. GAtest is applied in the normal fashion to the given trapezoidal space. Suppose $\mathrm{xp} = (xp_1, xp_2, \ldots, xp_n)$ is the semi-integer solution obtained and xp_j the corresponding non-integer element. Now, the following steps are repeated until the dependence problem is solved.

1. If at any stage we find that the lower and upper limits for dimension i are same then we propagate this value to dimensions $(i+1)$ to n, eliminating the occurrence of i^{th} index variable in the loop bounds.

2. If xp lies within the bounds determined by $lmax_i$ and $umin_i$ or l_j and u_j are constants then we can partition the space normally.

3. If Step 2 is not applicable then we choose an i such that l_i and u_i are constant. Now search is conducted in the spaces

$$\begin{array}{ll} l_1 \leq x_1 \leq u_1 & l_1 \leq x_1 \leq u_1 \\ \vdots & \vdots \\ l_i \leq x_i \leq \lfloor xp_i \rfloor \quad \text{and} & \lfloor xp_i \rfloor + 1 \leq x_i \leq u_i \\ \vdots & \vdots \\ l_n \leq x_n \leq u_n & l_n \leq x_n \leq u_n \end{array}$$

It should be noted that if the same semi-integer solution is obtained repeatedly in Step 3 then the performance of the algorithm degrades. Assuming that there are sufficiently many semi-integer solutions and a proper random number generator is being used, the probability of getting stuck at the same solution is quite low.

Notice that the triangular spaces corresponding to the following example can be treated as a limiting case of trapezoidal spaces.

```
DO I = 1, 10
    DO J = I, 10
        :
    ENDDO
ENDDO
```

Hence, the scheme as outlined in this section can be applied to triangular spaces as well. Since a convex polygonal space can be partitioned into triangular spaces, the GAtest may also be extended to obtain integer solutions of a diophantine equation for a given polygonal space.

3.4 Solution Enumeration

When Algorithm 3.1 predicts dependence, it may still be possible to restructure the loop-nest to exploit the partial parallelism. If the dependence distance ($= k$)

is constant, the iteration space $[1, \ldots, N]$ of a loop can be partitioned into sub-intervals $v_0 = [1, \ldots, k], v_1 = [k+1, \ldots, 2k], \ldots, v_r = [rk+1, \ldots, N]$. The iterations of each sub-interval v_j are free of the dependence sinks assuming that the iterations of $v_0, v_1, \ldots, v_{j-1}$ have been executed [14]. The iterations of a sub-interval v_j can be executed in parallel.

On the other hand, if the dependence distance is not constant then the minimum distance between all the instances of the dependences can be used to form the sub-intervals v_j. This scheme is conservative in the sense that it leaves some parallelism unexploited. A more powerful method is to block the maximum number of iterations together such that no group contains both source and sink of the dependence [14]. This scheme exploits the maximum parallelism possible. If the distance between the dependence takes the values $\alpha_1, \alpha_2, \alpha_3, \ldots$ then the sub-intervals of iterations can be formed as follows:

$$\begin{aligned} v_0 &= [1, \ldots, \alpha_1] \\ v_1 &= [\alpha_1 + 1, \ldots, \alpha_2] \\ \ldots &\quad \ldots \end{aligned}$$

GAtest is used to enumerate all the solutions and form the sub-intervals v_i to exploit the full parallelism of a given loop-nest in the following manner. The search space is partitioned successively such that the integer solution detected in a stage is excluded from consideration in the next stage. The GAtest is recursively applied to the the resulting subspaces this way until all the integer solutions to the given problem are detected.

More formally, the partitions may be obtained in the following manner. Let $\mathbf{xq} = (xq_1, xq_2, \ldots, xq_n)$ be the integer solution returned by GAtest within the

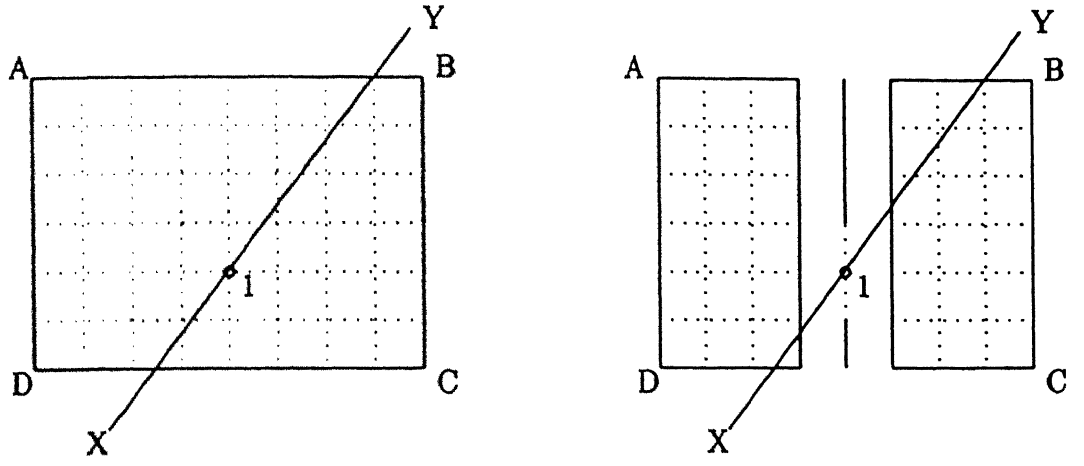


Figure 3.3: Solution enumeration using GAtest

bounds specified. For $k = 1, \ldots, n$ conduct the search in the following subspaces:

$$\begin{array}{ll} xq_1 \le x_1 \le xq_1 & xq_1 \le x_1 \le xq_1 \\ \vdots & \vdots \\ xq_{k-1} \le x_{k-1} \le xq_{k-1} & xq_{k-1} \le x_{k-1} \le xq_{k-1} \\ l_k \le x_k \le xq_k - 1 \quad \text{and} & xq_k + 1 \le x_k \le u_k \\ l_{k+1} \le x_{k+1} \le u_{k+1} & l_{k+1} \le x_{k+1} \le u_{k+1} \\ \vdots & \vdots \\ l_n \le x_n \le u_n & l_n \le x_n \le u_n \end{array}$$

The above scheme is applied recursively to the resulting spaces. We illustrate the idea in Figure 3.3 for the case of two dimensional space.

Since all the solutions can be enumerated using GAtest, it is possible to handle the coupled subscripts[2] as well. The trick is to apply the GAtest in the normal manner to each subscript for enumeration of all the solutions. The intersection of the solutions for each subscript gives the solution for the given pair of array references as a whole. For example, consider the following loop-nest:

[2]If the same index variable occurs in two different subscript expressions, the latter are called coupled subscripts [13].

```
N = 9
DO I = 1, N
    DO J = 2, N+1
        A(2*I+3*J+N, 4*I+J+N) = ...
        ...= A(I-J+N+3, 2*I-3*J+N-1)
    ENDDO
ENDDO
```

Applying the GAtest to the first subscript, the following solution set is obtained $P_1 = \{\langle 1,2,7,2\rangle, \langle 1,2,9,4\rangle, \langle 1,2,8,3\rangle, \langle 2,2,9,2\rangle\}$ for the tuple $\langle i,j,i',j'\rangle$. Similarly, for the second subscript, the GAtest provides the solution set $P_2 = \{\langle 1,2,8,3\rangle, \langle 1,3,7,2\rangle, \langle 1,4,9,3\rangle, \langle 1,5,8,2\rangle, \langle 1,7,9,2\rangle, \langle 2,3,9,2\rangle\}$. Now, $P_1 \cap P_2 = \{\langle 1,2,8,3\rangle\}$ gives the values of the index variables which satisfy both the subscript expressions simultaneously.

3.5 Dependence Direction

The solution enumeration is, obviously, a costly approach. It might, therefore, be useful to test for a given direction vector. When dependence does not exist for a particular direction vector then certain loop transformations may be applied safely, e. g., loop-interchange. Given the two statements S_1 and S_2 in a loop-nest, the GAtest can be used to determine if statement S_2 depends on S_1 with direction vector $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ in the following way.

Let $f(\mathbf{I})$ and $f'(\mathbf{I}')$ correspond to S_1 and S_2, respectively. We seek a solution to the following equation to solve the dependence problem:

$$f(\mathbf{I}) - f'(\mathbf{I}') = 0 \tag{3.2}$$

where $l_j \leq i_j, i'_j \leq u_j$ $(1 \leq j \leq n)$. Now, there are three cases possible depending on the value of s_j ("=", "<", ">").

"=": In this case, $i_j = i'_j$ and one of the variables (i_j or i'_j) may be eliminated from Equation 3.2.

"<": In this case, the objective is to search for a solution such that $i'_j < i_j$. The loop limits are modified to the following triangular space to constrain the search:

$$\begin{aligned} l_j &\leq i'_j \leq u_j - 1 \\ i'_j + 1 &\leq i_j \leq u_j \end{aligned}$$

">": Similar to the case for "<", the limits are changed as follows.

$$\begin{aligned} l_j &\leq i_j \leq u_j - 1 \\ i_j + 1 &\leq i'_j \leq u_j \end{aligned}$$

We apply GAtest to Equation 3.2 with the new constraints computed as given above. The presence of an integer solution implies a dependence for the given direction vector s.

Suppose that we wish to test whether or not there exists a dependence from statement S_1 to S_2 with direction vector s = (">", "=") in the following loop-nest:

```
DO I = 1, 10
    DO J = 1, 5
        S1 : A(3*I+2*J-5) = ...
        S2 : ...= A(4*I-J+2)
    ENDDO
ENDDO
```

Since s_1 =">" and s_2 ="=", the limits of i and i' get modified as indicated above and j' is eliminated. The resulting equation is

$$3i - 4i' + 3j = 7 \tag{3.3}$$

with the set of constraints

$$\begin{aligned} 1 &\leq i \leq 9 \\ i+1 &\leq i' \leq 10 \\ 1 &\leq j \leq 5 \end{aligned}$$

When the GAtest is applied to Equation 3.3 with the new limits, the existence of an integer solution (e. g., $(i = 1, i' = 2, j = 4)$) is indicated. Hence, dependence for the specified direction vector is concluded.

3.6 Implementation Issues

GAtest is based on genetic algorithms which greatly affect its behavior in terms of the following important metrics — the total number of iterations of the loop in Algorithm 2, the time taken and the variance of the time taken. Since GAtest is non-deterministic, the time taken by the different runs of the algorithm on the same input data may be different. To make the performance of GAtest more predictable it is desirable to reduce the variance of the time taken. In general, a tradeoff is observed between the number of iterations and the time taken. The reason is, while a complex operation helps in faster convergence, it also increases the time taken per iteration of the loop.

From the perspective of genetic algorithms, the process of finding a semi-integer solution in the search space consists of the following steps which are repeated until a semi-integer solution is obtained.

- Initialize the population.
- Apply some genetic operator to one or more of the vectors in the population.
- Select a population vector and optimize it.
- Use survival of the fittest to discard the worst fit vector.

The objective is to improve the average fitness of the population in the successive iterations. But at the same time, we should be careful not to get trapped in a locality in the search space which does not contain a solution. Although the probability of the presence of a solution is more for a fitter vector, it is also possible that various points in a locality have high fitness value but there is no solution in the vicinity. Hence, it is needed to incorporate some randomness in the algorithm.

The various genetic operators used which influence the various characteristics of GAtest are described below. The behavior of the algorithm has also been explained with the help of some graphs showing the relationship between the time taken and the variation in certain parameters. While the exact form of the characteristics would be problem dependent, the basic nature of the relationships is seen to be independent of the problem instance.

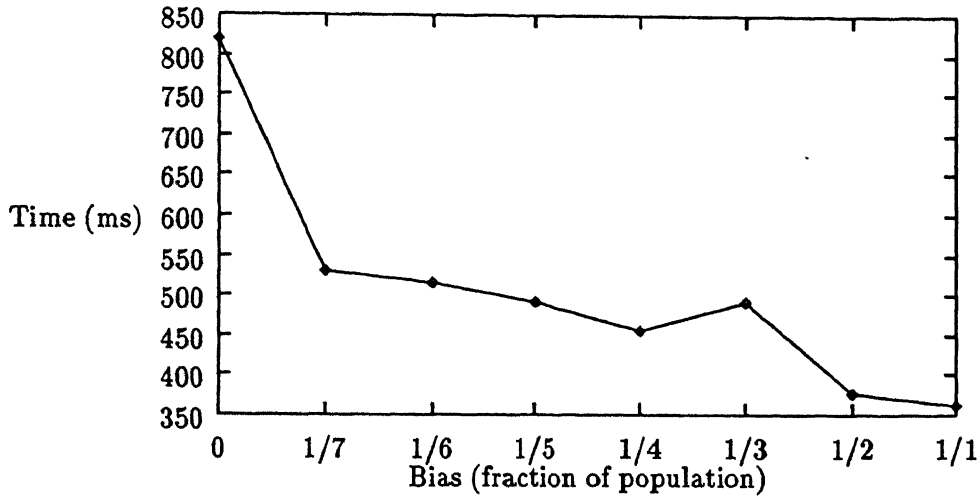


Figure 3.4: Effect of bias on convergence rate

3.6.1 Mutation

The GAtest employs a guided mutation strategy to optimize the selected vector by changing one of the elements x_j to $(a_0 - \sum_i^{i \neq j} a_i x_i)/a_j$ if the new value computed is within the loop bounds. In case the new value is not within the bounds, it is coerced to one of the loop bounds as given in the Algorithm 2. Following this strategy, if a semi-integer solution is detected randomly in the search space, it may result in high variance in the following way. Suppose the search space is broken up into two parts in such a way that one part is considerably larger than the other one and both the parts contain semi-integer solutions. If the smaller part is chosen first to obtain a semi-integer solution, it would generally take much less time compared to the case when the larger part is chosen first. This problem is circumvented by following a scheme which attempts to break up the search space in two almost equal parts. GAtest is applied to both the spaces and because they are of almost equal size, the time taken is almost same irrespective of the choice of the partition.

3.6.2 Biased Population

To obtain a semi-integer solution in the given search space, GAtest works with some integer vectors known as population. The elements of these vectors satisfy the

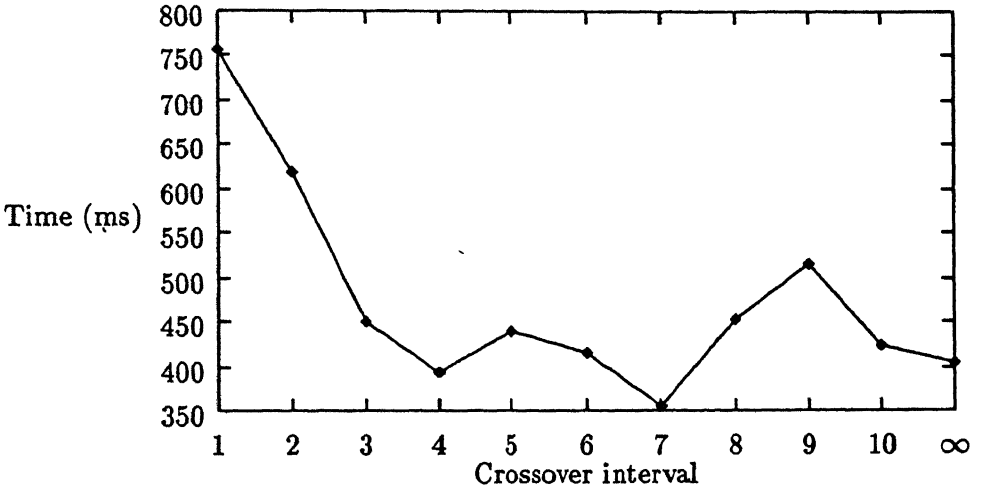


Figure 3.5: Effect of crossover on convergence rate

constraints imposed by the loop limits. If there is an integer solution of the given diophantine equation then a solution may be obtained more quickly if the vectors are initialized in the vicinity of the hyperplane representing the diophantine equation. This follows from the fact that the genetic algorithms employed follow a building block approach and new vectors result from perturbing a vector or combining two vectors.

Figure 3.4 depicts the relationship between the time taken and fraction of the population vectors that were biased. It may be noted that a high bias results in faster convergence. The problem instances which have one or more integer solution(s) display the characteristics shown in the figure. It is also likely that there is no integer solution to the given equation. In this case, initialization of the population with fitter vectors is not much of an advantage. Since the existence of a integer solution is not known in advance, it is useful to keep only a fraction of the vectors to be biased.

3.6.3 Crossover

The crossover operation randomly picks two vectors $\mathbf{X}$ and $\mathbf{Y}$ from the population, a site j $(1 \leq j \leq n)$ and then performs the crossover by interchanging the first j elements in the vectors $\mathbf{X}$ and $\mathbf{Y}$. The crossover operation results in new vectors with

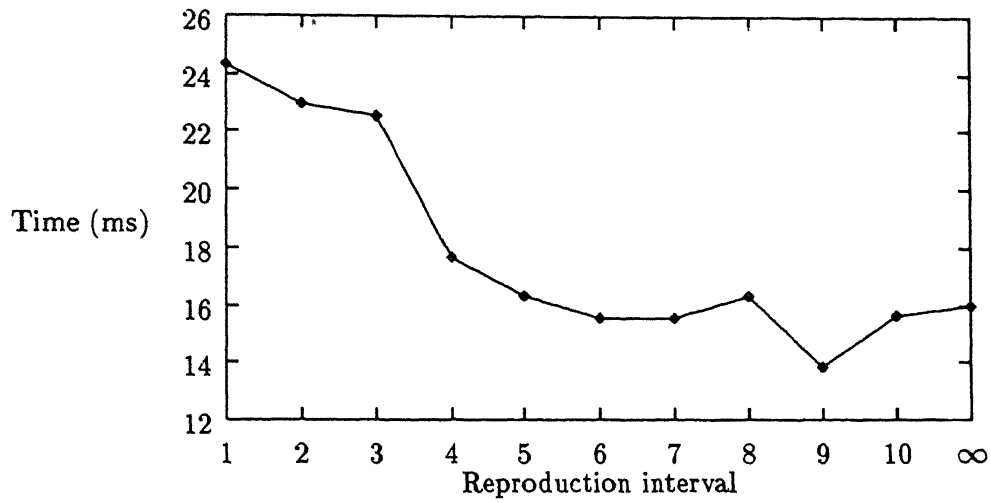


Figure 3.6: Effect of reproduction on convergence rate

high fitness value. The resulting set of new generation vectors helps in conducting the search in different directions. The crossover being a costly operation, it is likely increase the time taken for convergence although the algorithm may converge in fewer iterations. Figure 3.5 shows the relationship between the time taken and the frequency of applying crossover. As we increase the crossover interval, the overheads are reduced but after a particular point, the crossover does not have any significant effect on the rate of convergence. Rather, it takes more time to analyze the problem owing to the operations associated with crossover.

3.6.4 Reproduction

We try to keep more copies of the fitter vectors in the next generation by making the number of copies of a vector proportional to its fitness value. This operation is called reproduction. It improves the average fitness of the population and results in faster convergence. It can be argued that a vector may have very high fitness but it may not lead to the solution. If the reproduction is applied as described then we would have relatively more copies (maybe all) of this vector which may degrade the performance. To take care of this problem, the reproduction operator is damped artificially by bringing the fitness of the best and the worst vectors closer.

Since reproduction is also a costly operator, its characteristics are similar to

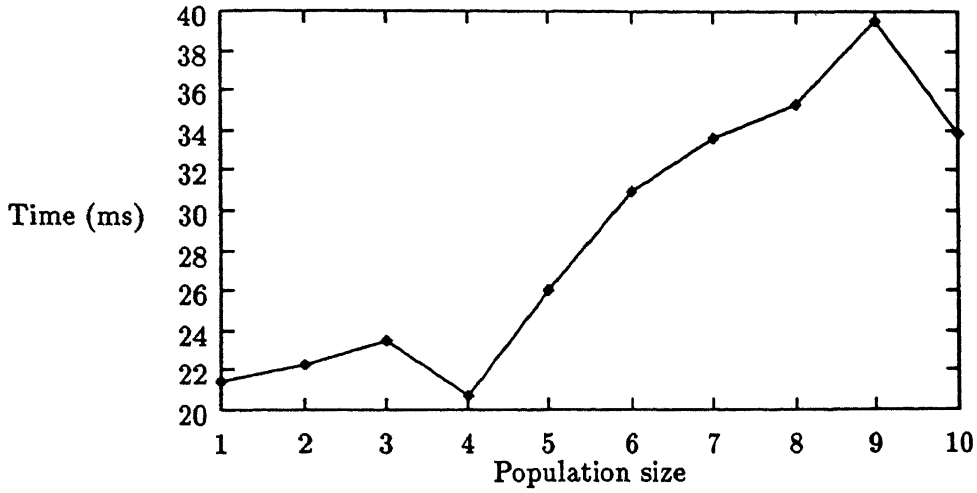


Figure 3.7: Effect of population size on convergence rate

that of crossover. We found that applying reproduction operator is not very useful because reproduction affects all the vectors of the population and the probability of it discarding some useful vectors is high. Figure 3.6 shows the characteristics of reproduction.

3.6.5 Population Size

The probability of some individual vector being close to a solution are higher with a larger population. Hence, the convergence is expected to be faster by using a large population size. But at the same time, a large population increases the cost of the various operations, e. g., finding out the worst fit vector. The optimal population size depends on other factors as well, for example, the nature of the given diophantine equation.

A typical graph of the population size versus the time taken by the algorithm is shown in the Figure 3.7. It is observed that the optimal population size is in the range of 4 to 6 for most of the benchmark programs.

3.7 Performance of GAtest

Let $C(B)$ and $C(\text{GAsemi-integer}(\mathbf{l}, \mathbf{u}))$ denote the cost of performing Banerjee's test and finding a semi-integer solution using Algorithm 2, respectively. Then, the cost $C(GA(\mathbf{l}, \mathbf{u}))$ of performing GAtest in the range $(\mathbf{l}, \mathbf{u})$ can be computed as follows.

$$C(GA(\mathbf{l}, \mathbf{u})) = \begin{cases} C(B) \text{ if Banerjee's test rules out the dependence} \\ C(B) + C(\text{GAsemi-integer}(\mathbf{l}, \mathbf{u})) + C(GA(\mathbf{l}, \mathbf{tu})) + C(GA(\mathbf{tl}, \mathbf{u})) \\ \quad \text{otherwise} \end{cases}$$

The bounds $\mathbf{tl}$ and $\mathbf{tu}$ are the new bounds corresponding to the non-integer solution obtained as outlined in Algorithm 2. Thus in the worst case, the GAtest would take $O(N(C(B) + C(\text{GAsemi-integer}(\mathbf{l}, \mathbf{u}))))$ time where N is the size of the search space. The complexity of the Banerjee's test $C(B)$ is $O(n)$ where n is the number of terms in the diophantine equation. $C(\text{GAsemi-integer}(\mathbf{l}, \mathbf{u}))$ depends on various factors as described in Section 3.6. The cost of finding a semi-integer solution may be optimized by using appropriate values of the different parameters.

In the average case, GAtest keeps on pruning the search space whenever the Banerjee's test is successful. Also, we need not consider region corresponding to $(\mathbf{tl}, \mathbf{u})$ if we get an integer solution in the space $(\mathbf{l}, \mathbf{tu})$. Since the execution time of the algorithm depends on a number of factors (e. g., the nature of diophantine equation, the size of the search space) statistical analysis is the only way to predict its behavior.

The GAtest was implemented in Wolfe's TINY tool [21] and extensive performance analysis was done to compare its performance with the Omega test. The Omega test was chosen for comparison because it is a well studied exact test of a general nature and its performance is known to be very good. Two versions of the GAtest were used — one which enumerates all the solutions and the other which gives only yes/no answer. On the other hand, the Omega test calculated the appropriate direction vectors. It also reported the distance vector for constant dependence distances. NASA NAS test suite and all the `tiny` source files supplied with TINY were used for benchmarking. We also tested the algorithms for some of our own programs. Table 3.1 gives the time taken by the Omega test, GAtest

Program	Search Space Size	Number of Solutions	Total time taken for analysis (in ms)		
			Omega	GA (yes/no)	GA (all)
vpenta	4.7×10^6	3.7×10^4	5108	793	74474
btrix	5.1×10^5	1.8×10^4	5030	1524	27596
ocean	1.5×10^3	1.3×10^2	92	20	1548
relax	9.6×10^4	9.8×10^2	74	10	1162
across	9.8×10^4	9.8×10^2	52	8	826
convex1	2.8×10^4	2.2×10^2	28	8	272

Table 3.1: Comparison of GAtest with Omega Test

with yes/no answer and GAtest with enumeration for some programs used for performance analysis. The analysis was done on Sun-3/60 workstation.

Since the GAtest cannot be applied for unknown loop bounds, we excluded such loops (for example, loops which make use of index arrays) from consideration. Furthermore, the values of dummy parameters of subroutines were made known to the dependence tests through user assertions. The choice of value of such parameters merely helps the GAtest to eliminate the same symbolic terms in the subscripts. In fact, none of such cases affected the outcome of the test for the programs listed in Table 3.1. The timings for GAtest were averaged over sufficient number of runs owing to its non-deterministic nature. The time taken by the different tests, as reported in Table 3.1 also includes the time taken to build the problem.

The results indicate that the GAtest with yes/no answer is considerably faster than the Omega test. While the solution enumeration, in general, takes more time than the Omega test, it can expose more parallelism, if the dependence distance is not constant. In such cases, the Omega test provides only the direction vector which is not much helpful in executing the different iterations in parallel. The time taken by the GAtest for solution enumeration depends on the search space size and the number of integer solutions of the given diophantine equation. If the search space is large and there are a number of integer solutions then the solution enumeration would be likely to take more time owing to less pruning of the search space.

The programs listed in Table 3.1, mostly implement some numerical algorithms. `vpenta` inverts three pentadiagonals simultaneously. This routine is commonly employed to solve systems of partial differential equations. `btrix` is a tridiagonal

solver. `relax` is relaxation algorithm for two dimensional grid. The other programs are small routines containing a sample of commonly encountered array references.

In Table 3.1, the search space size is specified by summing up the search space sizes corresponding to all the array references analyzed by the GAtest. Similarly, the number of solutions is obtained by summing up the number of solutions reported by the GAtest through subscript by subscript analysis of all the array references. It can be noted from Table 3.1 that the time taken for the enumeration is considerably more for `vpenta` than for `btrix`. It can be explained by observing that `vpenta` has more number of solutions and its search space size is also larger compared to `btrix`. Other programs also exhibit more or less similar behavior considering the fact that the time taken is also a function of the nature of the diophantine equations formed from the subscript expressions. The relationship between the search space size, the number of solutions and the time taken for enumeration is noted to be more uniform for randomly selected array references.

Although solution enumeration using the GAtest may prove costly, it is justified because of the fine grain parallelism that can be exploited. In fact, the extra compile time may be more than offset by the decrease in the execution time of a program on a multiprocessor. In the case of varying dependence distance, the solution enumeration can be used to execute the maximum possible iterations in parallel, as suggested in Section 3.4. The dependence distance for a given pair of array references is usually not constant when: (1) There are coupled subscripts. (2) The nonzero coefficients are different in the same subscript. An empirical study of array references in real programs reported that the frequency of dependences with non-constant distance is as high as 86.35% [17]. Out of these, all the linear subscripts can be handled by the GAtest resulting in improved loop parallelization.

At the other extreme, the GAtest with yes/no answer provides a fast exact test of general nature. It can replace other fast inexact methods which give highly conservative answers. We found that the GCD and Banerjee's tests (it also computed the possible direction vectors) took a total of 1875 ms to analyze the programs listed in Table 3.1. In comparison, the GAtest(yes/no) took 2363 ms which is just 26% more than the time taken by the GCD and Banerjee's tests. Since the GAtest can

also test a dependence relation for a given direction vector, it turns out that the loop restructuring can be done based on GAtest only.

The divide and conquer strategy used by GAtest lends itself to be easily parallelized. As the search space is partitioned, the various subspaces may be allocated to different processors which can work independently.

Chapter 4

Conclusions

The data dependence analysis is an important part of any restructuring compiler. The subject has received great attention from the researchers. As a result, there are a number of dependence tests available. Unfortunately, there is no exact test of general nature, which can provide the required information at a low execution cost. This work proposed a practical approach, the Zero test, to integrate all the important tests. The efficiency of the dependence analysis may be considerably increased by using this approach to selectively apply the dependence tests. The general scheme suggested in this thesis may also be tailored to suit the needs of particular applications.

This thesis presented the GAtest, a new method that solves the dependence problem for a given pair of array references. It is an exact test of general nature, yet it can detect the integer solutions at a low execution cost. The data dependence analysis in practical systems can be significantly improved by replacing other inexact tests, which often conclude dependence wrongly, by the GAtest. Since the GAtest can tackle trapezoidal and triangular loops and can provide the direction vector, the loop restructuring can be done based on the GAtest only.

None of the available dependence tests addresses the issue of efficiently extracting the fine grained parallelism by enumerating all the solutions. The GAtest follows the divide and conquer paradigm and hence, it can handle the solution enumeration by searching sub-spaces in a natural and efficient way. The solution enumeration is

more expensive than providing only direction and distance vector. But perceiving compilation as a one-time job, the high cost of the solution enumeration using the GAtest may be afforded, keeping in view the improved execution time. Some specific applications, which demand the maximum parallelism to be extracted, may derive a great benefit from the faster execution. In fact, the solution enumeration is the only way to get optimal performance if the dependence distance is not constant. It also helps the GAtest in handling the coupled subscripts.

The Zero test is a practical approach which can be modified to incorporate other new tests. A more extensive performance analysis of the various tests may considerably improve the efficiency of the Zero test. The array references may be classified according to their characteristics. A comparison of the performance of the different tests for each class may then be used to choose probabilistically more efficient tests for a dependence problem.

Although the GAtest addresses many of the important issues in practical dependence testing, some improvements are possible. The GAtest assumes dependence whenever the loop bounds are not known. Moreover, there is no cost effective way to calculate the distance vector. An enhancement of the GAtest to incorporate these features would increase its applicability and the efficiency of data dependence analysis.

Using genetic algorithms as a search technique is a new and exciting concept. We have discussed how it can be effectively applied to dependence analysis. It can also be extended to obtain a search strategy for semi-bounded spaces. Although exact information cannot be guaranteed in this case, a suitable choice of various genetic operators may improve the probability of obtaining solutions, if any. Since the semi-bounded spaces model `while` loops and `do` loops with symbolic variables, this extension would enable the GAtest to handle a new class of problems.

The GAtest, as presented in this thesis, employs very simple strategies to prune the search space. Some concepts from computational geometry and theory of linear diophantine equations may prove to be effective to prune the search space. Of course, there is a tradeoff between using simple, fast strategies and complex, expensive methods.

References

[1] U. Banerjee. *Dependence Analysis for Super Computing.* Kluwer Academic Publishers, 1988.

[2] H. Braun. On solving travelling salesman problems by genetic algorithms. *Lecture Notes in Computer Science*, 496:129–133, 1991.

[3] J. P. Cohoon, W. N. Martin, and D. S. Richards. Genetic algorithms and punctuated equilibria in VLSI. *Lecture Notes in Computer Science*, 496:134–141, 1991.

[4] G. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[5] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *ACM SIGPLAN '91 Conf. Programming Language design and Implementation*, pages 15–29, Toronto, Canada, June 1991.

[6] D. E. Goldberg. *Genetic Algorithms in Search, Optimzation and Machine Learning.* Addison-Wesley Publishing Co., Reading, MA, 1989.

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Mich., 1975.

[8] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw Hill Book Co., New York, 1984.

[9] D. E. Knuth. *The Art of Computer Programming, Semi-numerical Algorithms*, volume 2. Addison-Wesley, Reading, MA, second edition, 1981.

[10] X. Kong, D. Klappholz, and K. Psarris. The I test: A new test for subscript data dependence. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[11] D. J. Kuck et al. The effect of program restructuring, algorithm change and architecture choice on program performance. In *International Conference on Parallel Processing*, August 1984.

[12] Z. Li and P. Yew. Some results on exact data dependence analysis. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[13] Z. Li, P.-C. Yew, and C.-Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):26–34, 1990.

[14] C. D. Polychronopolous. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[15] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[16] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *ACM SIGPLAN '92 Conf. Programming Language design and Implementation*, pages 140–151, 1992.

[17] Z. Shen, Z. Li, and P.-C. Yew. An empirical study on array subscripts and data dependences. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[18] H. R. Sudheer. GAtest: An exact dependence test for loop parallelization. Master's thesis, I. I. T., Kanpur, 1993.

[19] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelizatioin of Call statements. In *Proceedings of SIGPLAN '86 Symp. Compiler Construction*, pages 176–185, Palo Alto, CA, June 1986.

[20] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[21] M. Wolfe. The Tiny loop restructuring research tool. In *Proceedings of 1991 International Conference on Parallel Processing*, 1991.

[22] M. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):591–601, 1992.